OASIS - An ASIS Secondary Library for Analyzing Object-Oriented Ada Code

Alexei Kuchumov[1], Sergey Rybin[1], Alfred Strohmeier [2]

[1] *Scientific Research Computer Center*
Moscow State University, Vorob'evi Gori
Moscow 119899, Russia
mailto:rybin@alex.srcc.msu.su

[2] *Swiss Federal Institute of Technology in Lausanne*
Software Engineering Lab, Department of Computer Science
1015 Lausanne EPFL, Switzerland
mailto:alfred.strohmeier@epfl.ch

Abstract: ASIS has proven to be an effective platform for developing various program analysis tools. However, in many cases ASIS, as defined in the ASIS ISO standard, appears to be at a very low-level of abstraction compared to the needs of the tool developer. Higher-level interfaces and common libraries for specific needs should therefore be developed. The paper describes a ASIS secondary library providing abstractions and queries for analyzing object-oriented Ada code.

Keywords: Ada, Ada Semantic Interface Specification, ASIS, Object-Oriented Programming, GNU Ada Compiler, GNAT, ASIS-for-GNAT.

1 Introduction

The Ada Semantic Interface Specification (ASIS) [1] [2] is an interface between an Ada environment [3] [4] and any tool or application requiring statically-determinable information from this environment.

The set of the basic ASIS abstractions closely corresponds to the basic notions used in the Ada Standard [4], called RM 95 for Reference Manual: an ASIS Context represents an Ada environment, an ASIS Compilation_Unit represents an Ada compilation unit, and an ASIS Element models a syntactic construct, e.g. a declaration, a statement, an expression, etc. [5]. Operations on these types and the results of their calls are called “queries” in ASIS terminology.

One of the primary ASIS design goals was to provide a minimal complete set of abstractions which would allow tools to get all the static syntax and semantic properties of Ada code. “Completeness” means that any syntax or semantic property defined in RM 95 should be either directly retrievable by some ASIS query or it should be possible to derive it from the results of other queries. “Minimal set” means that ASIS tries to avoid any duplication of functionality and that it provides directly only “basic” information about the Ada code, letting the tool compute “higher-level” properties of the Ada code.

For many tools, however, the set of abstractions and queries directly provided by ASIS is at a level too low. Tool developers therefore often create first a higher-level library based on the standard ASIS abstractions and that better suits the tool's needs. Such a

International Conference on Reliable Software Technologies - Ada-Europe 2001, Leuven, Belgium, May 14-18, 200, Dirk Craeynest, Alfred Strohmeier (Eds.), Lecture Notes in Computer Science, vol. 2043, Springer, pp. 113-122.

library is called an ASIS secondary library, and abstractions and queries defined in a secondary library are called secondary abstractions and secondary queries. The tool is finally built on top of ASIS and such a secondary ASIS library.

In [6] we presented the general idea of OASIS - the ASIS secondary library providing a set of abstractions for analyzing properties of object-oriented Ada code. This paper provides an overview of the design and implementation of OASIS and demonstrates how its use simplifies the development of tools performing object-oriented specific analysis of Ada components.

2 ASIS terminology and the object-oriented model

We will start by defining the terminology used in the rest of the paper.

2.1. ASIS terminology

Most of the ASIS terms we will need were already mentioned in the previous section. We briefly repeat them here, with some additions and extra precision:

- *Primary queries* and *abstractions* are subprograms and types defined in the ASIS standard; an abstraction represents some syntax or semantic notion.
- A *secondary query* is a query providing some useful syntax or semantic information from the Ada environment and implemented as a combination of primary and other secondary queries. A *secondary ASIS library* defines a set of secondary queries and *secondary abstractions*. For each secondary abstraction, there should be some mapping to primary ASIS abstractions. No secondary queries and abstractions are defined by the ASIS standard.

The set of primary ASIS abstractions includes:

- *Context* is the abstraction for an Ada compilation environment, as defined in RM 95, 10.1.4. In most cases, a Context can be viewed as a set of ASIS Compilation Units.
- *Compilation Unit* is the abstraction for an Ada compilation unit, as defined in RM 95.
- *Element* is the abstraction for a syntax construct of a (legal) Ada compilation unit. Besides some minor technical details, we can say that an ASIS Element can represent any syntax component as defined by the syntax of Ada in RM 95. An ASIS Element can also represent an implicit declaration, such as an inherited record component, or an inherited or predefined operation. Finally, Elements can also represent the results of generic instantiations.

2.2. Object-Oriented terminology

When using technical terms for Ada, we will conform to the RM 95.

Object-oriented terminology varies widely with the language, and Ada is not an exception. Because the Unified Modeling language UML [7] is an industry standard widely used for object-oriented analysis and design, we will use its terminology when comparing with Ada.

In UML, the most important basic concept is that of a class. A class groups together objects, called its instances, having common properties. Basically, there are two kinds of properties, attributes that encapsulate the state of an object, and operations that implement its behavior, including its interaction with other objects.

A class can be derived from another class by specialization, or inheritance; in this generalization-specialization relationship, the derived class is often called the child class, and the other class is called the parent class. The child class inherits the attributes and components of the parent class. It might add new attributes and operations. It might also redefine some operations.

In Ada 95, a class is realized by a tagged type or to a type extension together with its primitive operations. The attributes of the class are the components of the record type, and the operations are its primitive operations (sic!). To be a primitive operation or not is defined by special rules and syntactically they do not belong to the definition of the type. To the contrary of other object-oriented programming languages, like C++ and Java, Ada therefore does not have a specific syntax construct for a class (in the sense of UML). Inheritance is realized in Ada by type derivation. Instead of speaking of the child class, Ada will therefore say “the derived type”. Primitive operations are inherited by the derived type, and the implementation of such an operation can be redefined, or overridden as Ada says.

Ada 95 defines the notion of a (derivation) class (RM 95, 3.4.1(1)). This concept corresponds to the inheritance hierarchy rooted at some tagged type. A class-wide type in Ada is the way to denote such a hierarchy, but the hierarchy itself does not correspond to a specific syntax construct. The “closest” construct is the declaration of the tagged type or the type extension that is at the root of the hierarchy.

An important feature of object-oriented programming is dynamic binding of an operation to one of its implementations. The idea is that within an inheritance hierarchy an operation might have different implementations, and the right one is chosen only at run-time when the class, as UML would say, or the specific type, as Ada would say, of the object that executes the operation is known. In Ada, the choice between static and dynamic binding, the latter being called dispatching, is made when writing a call to the operation. This is in contrast to other languages: in Java, a call is always dynamically bound, and in C++, the decision is made when declaring the operation, i.e. the member function. To come back to Ada, dynamic binding only takes place when an actual parameter of the call is of a class-wide type.

Finally, Ada has the concept of a class-wide operation. One or several formal parameters of such an operation are of a class-wide type. When calling the operation, any actual belonging to the derivation class can be passed. Class-wide operations are not primitive operations, and therefore they are not inherited. They can be defined at any level of the inheritance hierarchy.

In the rest of the paper, we will use Ada's view on the object-oriented world. We will use the term *derivation item* to represent tagged type or a type extension together with all its components and all its primitive operations; a derivation item corresponds therefore to the concept of a class in UML terminology. We will use the term *derivation class* to represent a derivation hierarchy rooted at some derivation item.

3 Using ASIS for analyzing object-oriented properties of Ada code

ASIS' view of an Ada compilation unit is based on its syntax structure. All syntax constructs are mapped to so called ASIS Elements, and such an Element carries both the syntactic and the semantic properties of the corresponding Ada construct. Individual Elements are quite natural and sufficient when only syntactical information is needed. However the approach is very limited when it comes to semantic information. Indeed, by the very nature of the approach, a semantic property about some Element must be returned and represented by some *other* Element or set of Element(s). For example, if Element is an expression, a query about its type will return the Element representing the corresponding type declaration. Also, if Element represents a subprogram call, ASIS can inform about the called subprogram by providing the Element representing the declaration of the subprogram.

This “Element-based” approach works well when the tool needs only semantic information formulated for a single specific construct, e.g. the type of an expression, the definition of a name, the declaration of a called subprogram etc. But when the information of interest is about sets of Elements which are in some specific relation with each other, ASIS provides very limited capabilities, and the tool has to compute most of the information itself.

This is certainly the case for tools performing analysis of object-oriented Ada code.

Consider the following simple example of object-oriented Ada code:

```
package Pack1 is
   type A is tagged record
      Comp : Integer;
   end record;

   procedure P1 (X : A);
   procedure P2 (X : A);

   type A1 is new A with record
      Comp1 : Integer;
   end record;
end Pack1;

with Pack1; use Pack1;
package Pack2 is

   procedure P3 (X : A);

   type A2 is new A with record
      Comp2 : Integer;
   end record;

   procedure P1 (X : A2);

end Pack2;
```

From the point of view of ASIS, the code consists of two Compilation Units, both defining syntax Elements such as tagged type declarations, type extension declarations and subprogram declarations. When analyzing the code with ASIS primary queries, we can traverse its structure, either manually or by using some instantiation of the ASIS Traverse_Element generic procedure. By using the ASIS Element classification queries, we can easily detect the tagged types, record components, type extensions and subprograms.

In contrast, from the point of view of the object-oriented paradigm, the code defines a hierarchy of three classes, in classic object-oriented terminology, and a derivation hierarchy containing three derivation items in our terminology. Each derivation item has a position in the hierarchy, and a set of components and a set of primitive, inheritable operations. The derivation item at the root is the tagged type A with its derivable component Comp and the primitive operations Pack1.P1 and Pack1.P2; notice that Pack2.P3 is not a primitive operation of this type, and therefore does not belong to the derivation item.

To perform this kind of analysis of object-oriented code, an ASIS tool must first create abstractions for the concepts of a derivation item and a derivation class. It then has to use non-trivial combinations of standard “Element-based” queries to collect and to assemble all the needed information. Let's outline a possible approach, based on primary queries only.

A derivation item is linked to an ASIS Element representing the declaration of a tagged type or a type extension. The full set of components of a derivation item corresponding to a type extension is the union of the inherited components and of the components that are member of the extension part. The tool can retrieve the inherited components with the query Asis.Definitions.Implicit_Inherited_Declarations.

To get the set of operations of a derivation item, i.e. the set of primitive operations of the corresponding type, the tool must traverse the package declaration enclosing the corresponding type declaration. During this traversal, for each encountered subprogram declaration, including the implicit declarations of inherited subprograms, it must check if the subprogram is a primitive operation of the type. To find the derivation class rooted at some derivation item, e.g. the one associated with Pack1.A, the application must traverse (almost) the whole Context and collect all the derivation items related to types derived directly or indirectly from the root type.

4 Presentation of OASIS - an ASIS secondary library for object-oriented analysis

4.1. Design goals of OASIS

We suspect that most ASIS-based tools analyzing object-oriented Ada code share similar abstractions, i.e. data types and operations, to represent basic properties of such code. There is therefore a need for an ASIS secondary library defining and implementing such abstractions. For Ada programmers, usability of these abstractions is increased if they are based on Ada's view of object-oriented concepts. The OASIS project is an attempt to satisfy these needs.

Providing a set of high-level abstractions for the analysis of object-oriented Ada code was one design goal of OASIS. Another one was to integrate these abstractions with the “Element based” approach used in “core” ASIS. In other words, the goal was not to create a completely new model for extracting object-oriented information, but to extend in a natural way existing ASIS functionality.

The current version of OASIS is the result of a university research project. We are expecting some changes in the interface, such as repackaging and extending the functionality. But the first experiments of using OASIS have shown that OASIS is a great foundation for developing ASIS tools performing object-oriented analysis of Ada code.

4.2. Interface of OASIS

The main idea behind OASIS is to provide direct support for the main concepts of Ada's object-oriented programming concepts. The basic OASIS abstractions are the derivation item and derivation class.

An OASIS derivation item is related to the declaration of a tagged type or a type extension. In contrast to such a declaration, the basic properties of a derivation item include the complete lists of components and primitive operations of the corresponding type, together with the position of the type in the “enclosing” derivation class.

A derivation item associated with a tagged type is a root derivation item. Otherwise stated, a derivation item is called a root item, if it is not derived from some other derivation item. An OASIS derivation class is the hierarchy of all the OASIS derivation items that are derived directly or indirectly from a root derivation item.

OASIS defines its own specific abstractions to represent a basic property of a derivation item - a component or a primitive operation. These abstractions are based on the corresponding syntax constructs: a component declaration and a subprogram declaration. In OASIS, these abstractions have additional basic properties: a reference to the derivation item the component or primitive operation belongs to, and a reference to the derivation item that explicitly declares the component or operation.

To represent information about dynamic binding, OASIS defines two abstractions, one for a class-wide operation and one for a dispatching call.

OASIS is structured as a hierarchy of Ada packages rooted at the package named OASIS. The root OASIS package contains the definition of the types Derivation_Item, Component and Primitive. Each of the child packages contains type declarations and queries needed for one piece of the OASIS functionality: working with derivation classes, working with components, working with class-wide operations, etc. OASIS therefore follows ASIS' philosophy for packaging the interface.

The following list provides an overview of the functionality offered by the current version of OASIS:

- For each OASIS abstraction, there are queries for mapping OASIS concepts onto basic ASIS abstractions. It is therefore possible to “convert” an OASIS-defined entity into its corresponding ASIS Element, i.e. the syntax construct on which it is based, and to “convert” an ASIS Element into an OASIS entity, if any;

- It is possible to retrieve a list of all the root derivation items within a given construct. i.e. within an ASIS Element, within a given ASIS Compilation Unit or within a whole Context;
- For any root derivation item, it is possible to get the derivation class rooted at this item;
- For any derivation item, be it a root or not, it is possible to get the derivation class containing this item; the result can be limited to a given Element or a given Compilation Unit;
- For a derivation item, it is possible to get the list of all its ancestor items and the list of all descendants;
- For two derivation items, it is possible to find the nearest common ancestor item, if any;
- For a derivation item, it is possible to get the list of all its components and the list of all its primitive operations;
- It is possible to distinguish between “public” and “private” components, possible in Ada with private tagged types and private type extensions;
- For a primitive operation, it is possible to distinguish between an inherited and an explicitly declared operation;
- For a derivation item, it is possible to get the list of all the dispatching calls that could take this derivation item as an actual parameter;
- For a dispatching call, it is possible to get the list of primitive operations that can be invoked at run-time depending on the actual parameter;
- For a derivation item, it is possible to get the list of class-wide operations defined for it and for all its ancestors;
- OASIS defines its own generic list and tree data structures. Their instantiations are used for various OASIS abstractions, e.g. a derivation class is implemented as an instantiation of the generic tree. OASIS also defines traversal procedures for its generic lists and trees, used e.g. for traversing a derivation class.

4.3. Implementation of OASIS

From the very beginning, the OASIS project was based on the public version of the GNAT ASIS technology [8] [9] [10].

Two different implementation approaches can be considered: 1. Implement OASIS using only the standard ASIS functionality; 2. Implement it using the internal GNAT data structure, the Abstract Syntax Tree - AST. Choosing between these two approaches is not obvious - each of them having pros and cons.

Implementing OASIS directly on the AST simplifies the implementation of many OASIS queries, because semantic information is often available in the AST, but is not reflected by primary ASIS queries. E.g. in the AST, a boolean flag marks a controlling formal parameter in a subprogram specification. Also when implementing OASIS on top of standard ASIS, all kinds of unnecessary data structure transformations and checks must take place, leading eventually to poor performance. On the other side, working directly with the AST leads to a non portable implementation; also, and even when staying within the GNAT technology, the maintenance effort is higher since the structure of the AST may change with new GNAT versions.

Taking into account the research nature of the OASIS project and the fact that the main developer is not a specialist in the internals of GNAT, we chose to implement OASIS on top of ASIS, i.e. as a secondary library. The current version makes some use of the ASIS-for-GNAT specific Asis.Extensions package, which contains queries extending the standard ASIS functionality and is implemented directly on GNAT's internal data structures. However, the use of these non-portable extensions is quite limited, and only a temporary solution, we plan to get rid of in a near future.

We are not quite happy with the performance, but consider it as acceptable for a research project, especially since we have not yet investigated optimization possibilities. The OASIS-based sample tools (see Section 5) run quite fast on small Ada examples. The OASIS-based program that prints out all the derivation classes in a given Context takes several minutes to process the Booch components. This library consists in 146 Ada compilation units, contains more then 21'000 source lines of code, and defines 6 derivation classes that contain 99 derivation items altogether.

5 Examples of using OASIS

In this section, we will show how the use of OASIS simplifies the development of tools that analyze object-oriented Ada code.

5.1. Class browser

A class browser is a tool that allows one to navigate through class hierarchies, in the sense of basic object-oriented terminology. For Ada, class browsers are of special importance because the language does not have a specific syntax construct to represent a “class”, e.g. a data structure together with applicable operations. In OASIS, a Derivation Item fulfills this need, since it corresponds to the classic concept of a class. Furthermore, the OASIS query All_Roots returns the list of all root Derivation Items defined in either a Context, a Compilation Unit or an Element, depending on the actual parameter. A class browser could therefore start by extracting this list for the whole Context:

```
Root_List := All_Roots (My_Context);
```

For any Derivation Item, even if it is not a root item, it is possible to retrieve with the query Derivation_Class the transitive closure of the derivation hierarchy this Derivation Item is contained in. It is also possible to retrieve for any Derivation Item the subtree of which it is the root by calling the query Derivation_Subtree. To traverse a derivation hierarchy, the tool can use an instantiation of the generic traversal procedure Full_Traverse_Tree.

5.2. Detecting non overridden operations

In object-oriented code, not overriding an inherited operation is a potential source of errors, especially when new components are added to the derived type. For example, consider a hierarchy derived from the predefined type Ada.Finalization.Controlled; whenever a new component is added by a type extension, it is most likely that at least some of the primitive operations Initialize, Adjust and Finalize for this type should be overridden. We therefore think that a tool which detects non overridden operations in the presence of additional components might be useful.

The outline for such a tool is the following:

1. Get the list of all the derivation items used in the given program. This part is similar to the class browser problem.

2. Select all derivation items which are not root, for which there is at least one new component and for which at least one primitive operation is inherited “as is” from the direct ancestor. Compared with standard ASIS, this is easy to implement with OASIS, since we need exactly one OASIS query to get the complete list of components of a derivation item and exactly one OASIS query to get the complete list of its primitive operations.

3. Print out the selected derivation items together with the non overridden primitive operations.

5.3. Ideas for other tools

It might also be interesting to have a tool that finds all NON primitive operations of all tagged types and type extensions. Indeed, it happens quite often that a programmer thinks a subprogram is a primitive operation while it is not, because “placing” and freezing rules are not trivial in Ada.

A reverse-engineering tool that generates UML class diagrams out of object-oriented Ada code would also benefit from OASIS.

6 Conclusions

We think that the current version of the OASIS - the ASIS secondary library for getting the object-oriented-specific information about Ada code - is a good starting point for developing an industry-quality version of such a library. Even though OASIS is currently a university research prototype, first experiences in developing tools analyzing object-oriented properties of Ada code have shown that OASIS significantly simplifies the structure of the tools' code and cuts down the effort needed for their development.

References

[1] Ada Semantic Interface Specification (ASIS); International Standard ISO/IEC 15291 1999 (E).

[2] Currie Colket et all; Architecture of ASIS: A Tool to Support Code Analysis of Complex Systems; ACM Ada Letters, January 1997, vol. XVII, no.1, 1997, pp 35-40.

[3] John Barnes (Ed.); Ada 95 Rationale: The Language, The Standard Libraries; Lecture Notes in Computer Science, vol. 1247; Springer-Verlag, 1997; ISBN 3-540-63143-7.

[4] S. Tucker Taft, Robert A. Duff (Eds.); Ada 95 Reference Manual: Language and Standard Libraries, International Standard ISO/IEC 8652:1995(E); Lecture Notes in Computer Science, vol. 1246; Springer-Verlag, 1997; ISBN 3-540-63144-5.

[5] Sergey Rybin, Alfred Strohmeier; Ada and ASIS: Justification of Differences in Terminology and Mechanisms; Proceedings of TRI-Ada'96, Philadelphia, USA, December 3 - 7, 1996, pp 249-254.

[6] Sergey Rybin, Alfred Strohmeier, Vasiliy Fofanov, Alexei Kuchumov; ASIS-for-GNAT: A Report of Practical Experiences - Reliable Software Technologies - Ada-Europe'2000 Proceedings, Hubert B. Keller, Erhard Ploedereder (Eds), LNCS (Lecture Notes in Computer Science), vol. 1845, Springer-Verlag, 5th Ada-Europe International Conference, Potsdam, Germany, June 26-30, 2000, pp. 125-137.

[7] James Rumbaugh, Ivar Jacobson, Grady Booch; The Unified Modeling Language Reference Manual; Addison-Wesley, 1999.

[8] Sergey Rybin, Alfred Strohmeier, Eugene Zueff; ASIS for GNAT: Goals, Problems and Implementation Strategy; Proceedings of Ada-Europe'95, Toussaint (Ed.), LNCS (Lecture Notes in Computer Science) 1031, Springer, Frankfurt, Germany, October 2-6 1995, pp. 139-151.

[9] Sergey Rybin, Alfred Strohmeier, Alexei Kuchumov, Vasiliy Fofanov; ASIS for GNAT: From the Prototype to the Full Implementation; Reliable Software Technologies - Ada-Europe'96: Proceedings, Alfred Strohmeier (Ed.), LNCS (Lecture Notes in Computer Science), vol. 1088, Springer, Ada-Europe International Conference on Reliable Software Technologies, Montreux, Switzerland, June 10-14, 1996, pp. 298-311.

[10] Alfred Strohmeier, Vasiliy Fofanov, Sergey Rybin, Stéphane Barbey; Quality-for-ASIS: A Portable Testing Facility for ASIS; International Conference on Reliable Software Technologies - Ada-Europe'98, Uppsala, Sweden, June 2-8 1998, Lars Asplund (Ed.), LNCS (Lecture Notes in Computer Science), Springer-Verlag, 1998, pp. 163-175.